ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 1 (2012) 593–602

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Computational Science, ICCS 2010

DynaSched: a dynamic Web service scheduling and deployment framework for data-intensive Grid workflows

Shayan Shahand[1], Stephen J. Turner, Wentong Cai, Maryam Khademi H.

School of Computer Engineering, Nanyang Technological University, Singapore 639798

Abstract

Grid computing boosts productivity by maximizing resource utilization and simplifying access to resources which are shared among virtual organizations. Recently, the Grid and Web Service communities have established a set of common interests and requirements. The latest version of the Globus Toolkit implements the Web Service Resource Framework (WSRF) specifications which have been formulated to cover these interests. We leverage the Globus Toolkit to address some limitations in supporting the dynamic nature of large-scale Grid and data-intensive workflow executions.

Dynamic Web Service deployment fits well into the dynamic nature of the Grid and opens new ways of managing workflow executions on the Grid. In this article, we present the design and evaluation of a *dynamic Web Service scheduling and deployment framework (DynaSched)* that supports the workflow management of dynamic services. Dynamic Web Service deployment on the Grid allows jobs to be executed on the same site as where the input data is located. The empirical studies show that the designed framework decreases data-intensive workflow execution time by minimizing communication costs. We argue that the framework ensures more flexible, fault-tolerant workflows. The system is based on Open Grid Services Architecture specifications and is WSRF-compliant.

Keywords: Grid computing, Globus Toolkit 4, Web service, Dynamic deployment, Web service workflow, Data location awareness

1. Introduction and Motivation

During the last two decades, computing research was helping scientists apply distributed computing to challenging projects that pushed the limits of what could be done with conventional computing technology. Distributed computing technology has evolved into the Grid, a powerful computing environment to share geographically distributed, heterogeneous and dynamic resources among virtual organizations. The term resource in Grid computing is a general term to denote a computational, storage or instrument resource; however, for simplicity, we use the term resource in this article to refer to a computational resource. Data-intensive applications (e.g., experimental data analysis in which a job is executed on huge data sets, which could be distributed geographically) and compute-intensive applications (e.g., very large-scale simulation and analysis) seem more likely to benefit from the application of the Grid technology [1].

Workflow is a common technology for developing and executing such applications to harness distributed resources over the Grid. A workflow specifies a series of activities to be executed in a particular sequence. These workflows may either consist of conventional compiled executable jobs written in C/Java/etc. or be composed of Web Services (WSs). Workflows which are composed of Web Services are further referred to as *Web Service Workflows (WS-Workflows)* in this article. WS-Workflows are either *concrete* or *abstract*. In abstract workflows, jobs are not bound to specific

[1]Corresponding author. Contact: shahand@pmail.ntu.edu.sg

1877-0509
doi:10.1016/j.procs.2010.04.063

resources for execution while in a concrete workflows, jobs are bound to specific resources. Every abstract WS-Workflow must be converted to a concrete one before execution. This means that each job in the workflow should be bound to a particular Web Service which is deployed on a certain machine on the Grid.

The next generation of scientific workflows, if realized as WS-Workflows, can benefit from loosely coupled services[2], using a Service-Oriented Architecture (SOA), where every resource can be accessed as a service without concern about the underlying platform implementation. Thus, any workflow or user requirement can be addressed by these services [2]. Web Services can be used to develop an architecture according to SOA definitions.

Open Grid Services Architecture (OGSA) is a standard SOA based on Web Service concepts and technologies which assures interoperability of heterogeneous Grid resources [3, 4, 5]. Recently the Grid community and the Web Service community have established a set of common interests and requirements. A set of specifications, known as Web Services Resource Framework (WSRF), has been formulated to cover those requirements and realize OGSA [1]. Stateful Web Services are required by the OGSA definition. Web Services are stateless, i.e., they retain no data between invocations. WSRF provides a set of operations by which stateful Web Services can be implemented [6]. The latest release of Globus Toolkit Ver.4 (GT4), an open source software toolkit for building Grids, implements WSRF to meet the requirements of OGSA [7]. GT4 implements high-level services which provide essential functionalities of a Grid middleware according to the OGSA specification. Globus Toolkit is a de facto standard for Grid computing. Grid-based applications are developed on top of the GT4 high-level services [8]. Our framework uses these services and adds further services to enable dynamic deployment of Web Services on the Grid.

The converging specifications of the Grid community and the Web Service community reveal an efficient, flexible and powerful Grid environment. Loosely coupled Web Services are being used to implement the SOA of the GT4, which assures interoperability between heterogeneous systems (i.e., different types of resources can communicate and share information). WS-Workflow can be composed of a number of available Web Services provided by different contributors. A scientist can reuse existing Web Services or WS-Workflows to accelerate composing his own WS-Workflow. This could ease the development of service oriented Grid computing applications.

So far there is little work on scheduling dynamic Web Services on available computing resources, and the existing approaches do not use standard specifications like OGSA and WSRF (see Section 2 for more information). Deploying Web Services dynamically on available computing resources provides important advantages such as load balancing, minimum input/output data movement, high availability of service, reliability, fault tolerance, flexibility and efficient resource utilization. In this article we describe the design of a set of components that form an environment which is able to schedule abstract WS-Workflow execution on the Grid resources. Web Service jobs of the WS-Workflow can be deployed automatically and dynamically on Grid resources with the help of these components. We focus on data-intensive WS-Workflows to evaluate our framework.

The rest of this article is organized as follows: In Section 2 we provide a brief overview of related work. Section 3 presents the overall architecture of Dynamic Web Service Scheduling and Deployment Framework (DynaSched), functionality of its components and how they collaborate with each other. Next, in Section 4 we describe the evaluation and discuss the advantages of the framework. Section 5 concludes the article with a summary and recommendations for future work.

2. Related Work

A workflow management system is essential to execute workflows on Grid resources, using their computational power, storage capacity and instrument facilities. The workflow management system interacts with the Grid middleware (which is located between the Application level and System level — e.g., Globus Toolkit) to schedule, execute and monitor jobs on the Grid. There are many workflow management systems available from both the industrial and academic world. Each of these systems tries to optimize Grid resource utilization and enhance Grid application efficiency. Most of the existing workflow management systems execute conventional workflows which are composed of compiled executables written in C/Java/etc. (e.g., Condor [9], Pegasus [10], Triana [11], etc.). There are a few known workflow management systems which execute WS-Workflows (e.g., Taverna [12], Triana [11], Kepler [13], Askalon [14] etc.). However these systems do not support dynamic Web Service deployment. Indeed the only known workflow management system which uses dynamic Web Service deployment at the time of writing this article is ServiceGlobe [15].

ServiceGlobe is a lightweight infrastructure acting as a distributed and extensible service platform. ServiceGlobe provides dynamic service selection and invocation at runtime. Service selection is performed according to the technical specification of the desired service using UDDI. A generic, modular dispatcher service provides load balancing and high availability of services. This dispatcher replicates new services on idle resources [15].

Runtime loading allows distribution and replication of dynamic services on arbitrary resources in ServiceGlobe. Although this feature opens up a great optimization potential for ServiceGlobe in terms of load balancing, high availability and parallelism, there are a number of important issues in this system that are not addressed: The internal

[2]For simplicity we use the terms service and Web Service interchangeably throughout this article.

services which can be loaded at runtime must be implemented in Java using the ServiceGlobe API which confines the set of available services. In addition, in ServiceGlobe, dynamic services are stateless which is not desirable for most Grid applications. In this system, Web Services are loaded on any arbitrary resources according to the resource's workload and without concern about input data location. ServiceGlobe is a standalone project and does not rely on any Grid middleware infrastructure. Using standard Grid middleware infrastructures and certain standards eases secure and controlled resource sharing between virtual organizations. The ServiceGlobe platform project has not been active since 2003.

3. Framework Design

DynaSched is a modular framework which facilitates dynamic Web Service deployment on Grid resources. This framework consists of various components. Most of the framework components described in this section are Web Services themselves and are implemented in Java (i.e., they are Java Web Services using the GT4's Java WS Core API). Therefore, these components are residing in GT4's WS containers (Web Services must be deployed in a WS container to exploit them. The WS container is located on a resource and is responsible for marshalling, execution and de-marshalling of Web Service invocations [16]). The framework also uses some of the GT4 components such as GridFTP, WS-Monitoring and Discovering System (MDS), Grid Security Infrastructure (GSI) Java, and Java WS Core. GridFTP provides high-performance, secure, reliable data transfer for high-bandwidth wide-area networks. GSI authentication and authorization are used to ensure that only authorized parties can invoke the operations provided by the framework components.

To simplify understanding of the framework design, we divide the framework into three layers. The bottom layer, which is further referred to as the supporting components layer is a set of components on which the functionality of the Web Service scheduler relies. On the two higher levels, the Web Service scheduler and the WS-Workflow orchestration engine are located respectively. Figure 1 illustrates the DynaSched architecture and its components.

Different components are executed on different type of computational resources. There are three types of computational resources assumed: (i) *User machine*: The machine to which the user has direct access. The WS-Workflow Orchestration Engine (WS-Orc) is usually running on this machine. (ii) *Head-node*: A resource which is accessible by the user and primarily responsible for receiving and scheduling user jobs on compute-nodes. This resource is usually the gateway to the resources on which the Web Services are being executed. The Web Service Scheduler (WS-Scheduler) and the Web Service Code Repository (WS-CodeR) are usually running on this type of resource. (iii) *Compute-node*: Any resource which is not typically accessible directly by the user. Compute-nodes provide the computing power for the Web Services to be executed. The Dynamic Web Service Factories (WS-Factories) are usually running on these resources.

In the following sections we describe the functionality of the above mentioned components and explain the interactions between them. We follow a bottom-up approach in describing the layers of the framework.

3.1. Supporting Components Layer

This layer consists of a number of components which provide functionality that is required by the framework's main component (i.e., the WS-Scheduler).

3.1.1. Dynamic Web Service Factory (WS-Factory)

A WS-Factory is a Web Service itself and is deployed on each compute-node's WS container. WS-Factory deploys Web Services and undeploys them when they are not needed anymore. Qi et al. [17] proposed, (partially) implemented and evaluated the Highly Available dyNamic Deployment infrastructure (HAND) based on the Java Web Services core of GT4. This service (which is called `DeployService`) is shipped with the Globus Toolkit Version 4.2. We re-implemented the `DeployService` to add one further feature to it. WS-Factory is this reimplementation. First we look into details of `DeployService` and then we identify the feature which is added to it by WS-Factory.

Qi et al. [17] explored two different approaches for dynamic Web Service deployment: (i) *Container-level deployment* in which dynamic deployment of new services requires the reloading of the whole WS container. During the dynamic deployment and the WS container reloading process, all other services hosted by the WS container are unavailable. (ii) *Service-level deployment* in which no existing services are deactivated before deploying new services, assuming that the existing services are not redeployed. In the case of redeployment, the corresponding services are deactivated and re-activated after installation of the new version of services. This approach does not require the reloading of the whole WS container. All other services are unaffected and available in this approach.

The study by Qi et al. showed that container-level deployment works well when a global upgrade or configuration is needed, while service-level deployment is more flexible, capable and available. In addition the capability of dynamic deployment at the container-level is unpredictable in a complicated dynamic Grid environment [17]. However, at the time of writing, service-level deployment is not supported by GT4. Implementing this approach is complicated and requires low-level changes to the WS container. Therefore we used container-level deployment in the current version of the framework.

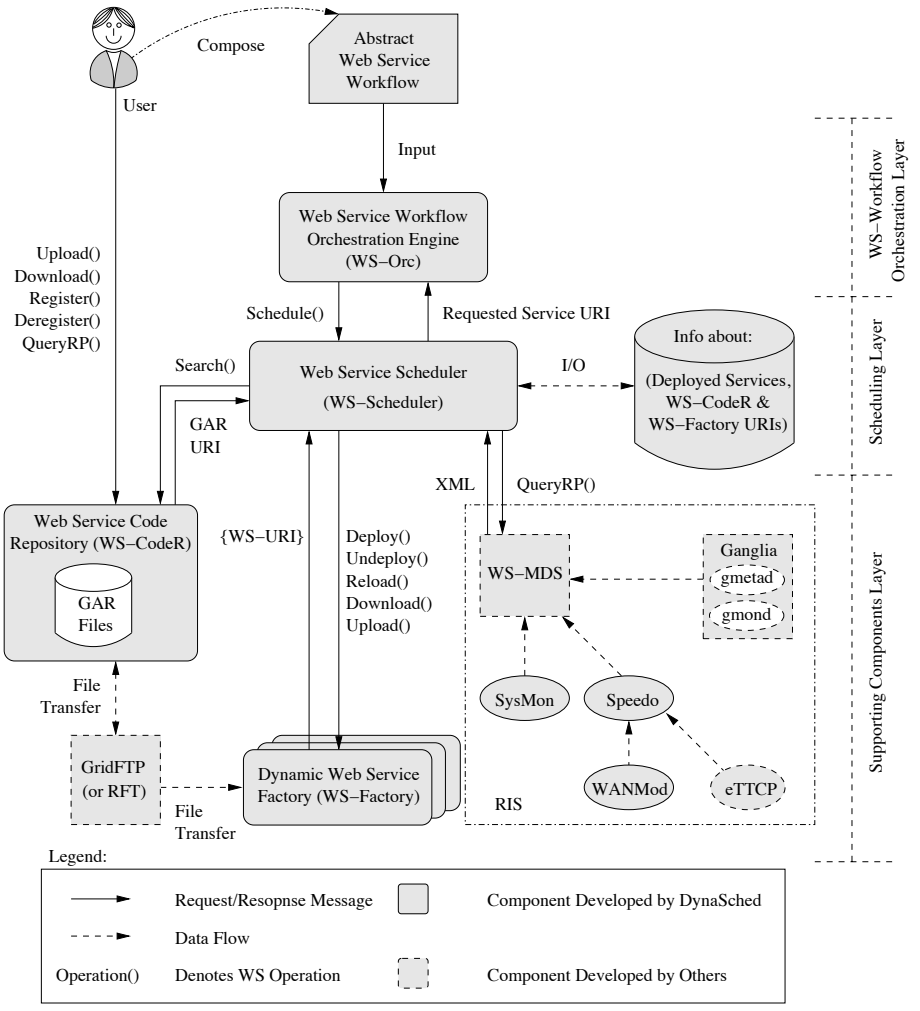


Figure 1: DynaSched architecture

The resource properties published and operations provided by the WS-Factory are identical to those of the `DeployService`. There are two operations for file transmission to the WS-Factory: `upload()` which uses SOAP attachments and `download()` which uses GridFTP to download the file from a given Uniform Resource Identifier (URI). In addition, three managing operations namely `deploy()` and `undeploy()` for dynamic Web Service deployment and undeployment, and `reload()` for reloading the WS container are provided. The only difference between `DeployService` and WS-Factory is that the `deploy()` function provided by WS-Factory also returns the dynamically deployed service URI. This functionality is required by the framework because the URI of the dynamically deployed service is used later to invoke its operations.

3.1.2. Web Service Code Repository (WS-CodeR)

WS-CodeR stores the undeployed WSs. Undeployed WSs are transfered to and registered in the code repository by invoking corresponding WS-CodeR operations. The WS-CodeR supports access control mechanisms, i.e., every registered undeployed WS has a specific owner, group and access attributes. The WS-CodeR publishes the list of registered undeployed WSs as a resource property. Therefore, any other service can search in it using the *QueryResourceProperties* interface (denoted as `QueryRP()` in figures).

WS-CodeR publishes one resource property, `GarList`, which contains the identifiers of the registered undeployed WSs in the WS-CodeR. There are two operations provided for file transmission to the WS-CodeR: `upload()` and

`download()` which are similar to those provided by WS-Factory. In addition, two managing operations namely `register()` and `deregister()` are provided to register and deregister an undeployed WS into/from the WS-CodeR. `Search()` operation is provided to search the WS-CodeR for a specific undeployed WS.

3.1.3. Resource Information System (RIS)

It is essential for the WS-Scheduler to have up-to-date information about the available resources. This information is provided by the Resource Information System (RIS) components. Information about the underlying Grid resources is collected by the MDS information services. We describe the GT4's information services and the information providers which are used in RIS here.

- *GT4 Information Services*: WS-MDS is a WSRF-compliant suite of Web Services to monitor and discover resources and services on Grids. The services which are used by RIS are: (i) Index service: Collects monitoring and discovery information from Grid resources, and publishes it in a single location. (ii) Aggregator framework: Collects data from an aggregator source and sends that data to an aggregator sink for processing. Aggregator sources include Execution source. Aggregator sinks include modules that implement the Index interfaces (e.g., GT4's `DefaultIndexService`). (iii) Execution source: Any program which generates an XML file can be configured as an execution source.
  To summarize, the execution sources generate XML files which contain the information about the resources on which they are located. These XML files are parsed by the Aggregator framework and published into the local Index service. The information published on the local Index services of each resource are aggregated and published on the head-node's Index service [18]. The information published by Index service is searched by the WS-Scheduler by using the *QueryResourceProperties* interface.

- *Ganglia*: Ganglia is a scalable distributed monitoring system for high-performance computing systems such as clusters and Grids. It is based on a hierarchical design targeted at federations of clusters. Ganglia components monitor changes in resource states and provide a number of static metrics (e.g., number of CPUs, CPU speed, total memory) and variable ones (e.g., workload, memory usage) [19]. The information provided by Ganglia is aggregated in the head-node and published on its Index service periodically.

- *SysMon*: SysMon is implemented as a scalable distributed monitoring system for Globus Grids. SysMon provides detailed information about CPU and memory load for every compute-node. The information provided by SysMon is published into the local Index service periodically and includes: (i) number of CPU cores, user/system/idle percentage for every CPU core, and (ii) used and available memory size. This component is a simplified alternative for Ganglia.

- *Speedo*: Speedo is implemented to provide a network connection bandwidth map in the form of a complete directed graph. It measures upstream and downstream bandwidth by using enhanced TTCP [20]. Speedo refreshes the information about the available bandwidth between resources in a regular basis. It transfers small packets to estimate the available bandwidth; therefore every bandwidth estimation task takes only a little bit of time. The information is generated in every compute-node and published into the local Index service periodically. The information includes: (i) resource name and its IP address, and (ii) list of the remote resources together with available bandwidths to and from them. To perform experiments in an emulated WAN environment, we developed a WAN model which replaces enhanced TTCP. We describe this model in Section 4.2.

3.2. Scheduling Layer

The scheduling layer contains two components: the WS-Scheduler and the framework database. The WS-Scheduler is the most important component of the framework because it is the main control of the framework. The WS-Scheduler uses the information provided by the RIS to schedule a dynamic Web Service deployment on the best available resource. It manages the WS components in the supporting components layer to perform the required corresponding actions.

3.2.1. Web Service Scheduler (WS-Scheduler)

A scheduler decides where and when to run a job. Dynamic Web Service deployment allows distribution, replication and relocation of Web Services to enhance productivity by maximizing resource utilization and minimizing execution time. This goal can be realized by deploying data-intensive Web Services (i.e., services which have large file(s) as their input/output) on or close to compute-nodes where the input/output data is located.

The WS-Scheduler decides which resource is the best possible option to use for a Web Service job. It uses a scheduling algorithm based on the characteristics of the Web Service to select the best available resource on which the Web Service can be deployed and invoked. After selecting the resource, if the requested Web Service is already deployed on that resource, the URI of the Web Service on that resource is returned by the WS-Scheduler. Otherwise if the requested Web Service is not deployed on the selected resource, the undeployed Web Service is transferred

from the WS-CodeR to the WS-Factory and deployed dynamically on the selected resource, then the URI of the dynamically deployed Web Service is returned by the WS-Scheduler.

For data-intensive Web Services, an input data location aware algorithm is implemented to select the best available resource. In this scheme, the WS-Scheduler selects a compute-node which is the same as or close to the resource where the input data is located. For data-intensive Web Services the execution time is trivial in comparison to the communication time (i.e., the time required to transfer the input file to the local host). Therefore the WS-Scheduler estimates the communication time and the time required for dynamic deployment and selects the resource with the least estimated time cost. The WS-Scheduler is a modular and customizable component. New scheduling algorithms can be implemented and plugged into the WS-Scheduler easily.

The input data location aware algorithm estimates the required time to execute a data-intensive Web Service dynamically for all available WS-Factories and selects the resource with the least time cost. The estimated time cost for executing a dynamic Web Service on WS-Factory$_i$ is denoted as t_i. Parameter t_i consists of a dynamic deployment time cost t_{id}, input data transmission time cost t_{it}, and Web Service execution time t_{ie}. Because of the characteristic of data-intensive Web Services, $t_{ie} << t_{it}$. Moreover, estimating t_{ie} is not easy. Therefore, t_{ie} is not considered in the time cost estimation formula. Typically for a given environment, t_{id} is roughly a constant value which depends on the WS container configuration, underlying Grid resource specifications and undeployed Web Service size. In the algorithm we show this constant value by parameter C. The most dominating factor in the time cost estimation formula for data-intensive Web Services is t_{it} which is calculated by dividing the input data size, denoted as S_{in}, by the measured available bandwidth between the resource on which the input data is located and the WS-Factory$_i$, denoted as BW_{in-i}. Note that if the input file is located on the same resource as where the WS-Factory$_i$ is deployed then $t_{it} = 0$. Once the time cost is estimated for all available WS-Factories the one with the least time cost is selected. If the requested Web Service is already deployed on the selected resource the URI of that service returned otherwise the undeployed Web Service is transferred to that resource, the `deploy()` operation is invoked and the resulted URI is returned.

The resource properties published by WS-Scheduler are as follows: (i) `WSCRList`: List of WS-CodeRs registered in the WS-Scheduler. (ii) `DWSFList`: List of WS-Factories registered in the WS-Scheduler. (iii) `DeployedServices`: List of dynamically deployed Web Services in the framework. (iv) `RequestQueues`: Status of the WS-Scheduler's deploy request queues.

The main operation provided by WS-Scheduler is the `schedule()` operation. This operation selects the best available resource for a requested Web Service based on the scheduling options and the information from RIS. If the selected resource has the requested Web Service already deployed, the URI of the requested WS on that resource is returned immediately. Otherwise it manages the dynamic Web Service deployment procedure and returns the URI of the dynamically deployed requested Web Service. The request message of this operation contains the requested WS identifier together with scheduling options data structure. In the scheduling options data structure, the preferred scheduling mode (e.g., data location aware) is specified and required information (e.g., input data location and size if using data location aware scheduling mode) is provided.

In addition to the `schedule()` operation, five managing operations are provided to clean up the framework (undeploy all dynamically deployed WSs) and register/deregister WS-CodeRs and WS-Factories. These operations are: `cleanup()`, `registerWSCR()`, `deregisterWSCR()`, `registerDWSF()` and `deregisterDWSF()` respectively.

3.2.2. Framework Database

The framework database stores the following information: (i) *Deployed Web Services*: Any Web Service which is deployed dynamically by the framework is registered in the database. (ii) *WS-CodeRs*: The framework has one or multiple WS-CodeRs. Every WS-CodeR is identified by an ID which is stored together with its URI in the database. (iii) *WS-Factories*: The framework has one or multiple WS-Factories. Every WS-Factory is identified by an ID which is stored together with its URI in the database. The framework supports multiple WS-Factories on multiple WS containers on a single compute-node.

The WS-Scheduler implements an interface to the database and the information stored in it is published as resource properties. Therefore any other service or Web application can query this information using the *QueryResourceProperties* interface.

3.3. WS-Workflow Orchestration Layer

This layer consists of one component which is described in the following section. Unlike other components of the framework (except WS-CodeR), the user interacts with this component directly.

3.3.1. WS-Workflow Orchestration Engine (WS-Orc)

To use the framework, a WS-Workflow orchestration engine is required which can convert an abstract WS-Workflow to a concrete one. The engine needs to interact with the scheduler during the conversion process. Currently there is no commonly agreed standard for describing abstract WS-Workflows. In order to test the underlying framework which is the main focus of this project, a customized workflow engine with only essential features is developed and used.

Currently the WS-Orc uses a command line GUI. The input of the WS-Orc is an abstract WS-Workflow where each Web Service is described by a Web Service identifier. WS-Orc manages the control flow and data flow for workflow execution. When it converts the abstract WS-Workflow to an executable one, it binds an abstract Web Service to a new or existing deployed Web Service by sending a scheduling request to the WS-Scheduler's `schedule()` operation. If a `schedule()` request times out, the WS-Orc retries by sending a new request. The `schedule()` operation returns a Web Service URI. The URI points to the location where the requested Web Service is deployed. WS-Orc can now invoke the requested Web Service.

4. Empirical Studies

A comprehensive evaluation of the effectiveness of the framework for WS-Workflows is challenging because of the difficulty of capturing the complexities of a realistic Grid environment and a variety of WS-Workflows. Therefore, we focus on specific experiments designed to show the effectiveness of the WS-Scheduler (and therefore the framework) for data-intensive WS-Workflows. We measure and compare the time required to execute WS-Workflows when using static services and container-level dynamic services. In the case of using static services (i.e., executing a concrete WS-Workflow), the WS-Orc invokes each Web Service directly. On the other hand, in the case of using dynamic services (i.e., executing an abstract WS-Workflow), the WS-Orc first converts each abstract Web Service to a concrete one by sending a `schedule()` request to the WS-Scheduler and then uses the corresponding URI returned by the WS-Scheduler to invoke the Web Service. The services which are dynamically deployed using the container-level approach are referred to as container-level dynamic services.

4.1. Experimental Environment

Our testbed consists of eleven resources connected by Myrinet. Each resource is powered by 3 GHz Intel Xeon dual-core processors and 1 GB memory running Linux 2.4.21 and JDK 1.5.0. We use the Globus Toolkit version 4.1.2. The GT4's WS container is executed on each resource and hosts all of the default services shipped with the toolkit. In addition to the default services the DynaSched's WS components, WS-Scheduler and WS-CodeR are hosted on the head-node's WS container and WS-Factories are hosted on the compute-nodes' WS containers. The number of nodes used in the experiments was limited by the availability of the Grid nodes in the testbed. With more Grid nodes available, we should be able to conduct relatively larger-scaled experiments. Throughout the experiments we have blocked access to these resources by other users in order to have a dedicated environment.

4.2. WAN Model

Data-intensive WS-Workflows are composed of data-intensive Web Services. The input data of data-intensive Web Services are typically distributed geographically on the Grid. In order to provide a realistic Grid environment for data-intensive WS-Workflow execution time measurements, we need a WAN emulator. We collected the data provided by S^3 [21], a scalable sensing service for large networked systems, to build a realistic WAN model based on real data. S^3 provides a snapshot of all-pair capacity and available bandwidth metrics updated about every 4 hours on PlanetLab. PlanetLab is a global research network that supports the development of new network services [22].

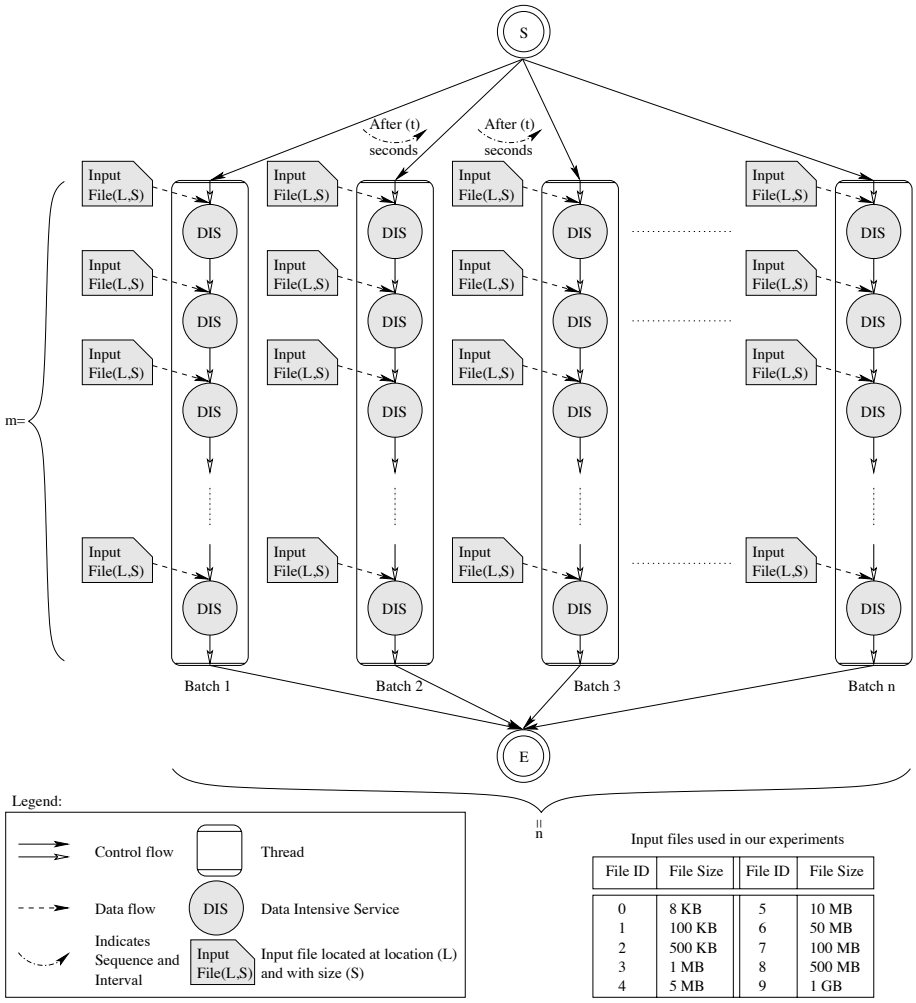
We use two terms to describe our WAN model: *connection* and *path*. Every connection is identified by its source node, destination node, and timestamp. Every path is identified by its source node and destination node. We collected available bandwidth for 6,090,278 connections during 14 days. Mean and standard deviation for 208,242 unique paths were calculated. After removing outliers, from the 151,921 remaining paths, 52,669 bidirectional paths were identified from which 37,672 of them were paths between 461 nodes located in different countries. A fully connected path graph with 10 nodes as the vertices and bidirectional paths as edges is extracted from the bidirectional paths model. Every bidirectional path in the model has two pairs of information: mean and standard deviation of the available bandwidth between its source to destination and vice versa.

The WAN model is used throughout the experiments performed to measure data-intensive WS-Workflow execution time. The WAN model replaces the enhanced TTCP component and provides Speedo with the emulated available bandwidth between real local Grid nodes. The WAN model is also used by the synthetic data-intensive Web Service to emulate the time required for input data transmission. Every time a component (i.e., Speedo or synthetic data-intensive Web Services) inquires the WAN model about the available bandwidth between two specific nodes, the WAN model generates a normally distributed random number according to the corresponding mean and standard deviation for that specific path.

4.3. Data Intensive Workflow

A data-intensive WS-Workflow is a workflow composed of data-intensive services. We implemented a synthetic data-intensive Web Service which accepts the URI of an input file, downloads the file and calculates its MD5 hash code.

As illustrated in Figure 2, each data-intensive WS-Workflow is composed of n batches and each batch has m invocations of the synthetic data-intensive service inside it (here $m = 10$). In the current section, wherever we refer to a



Input files used in our experiments

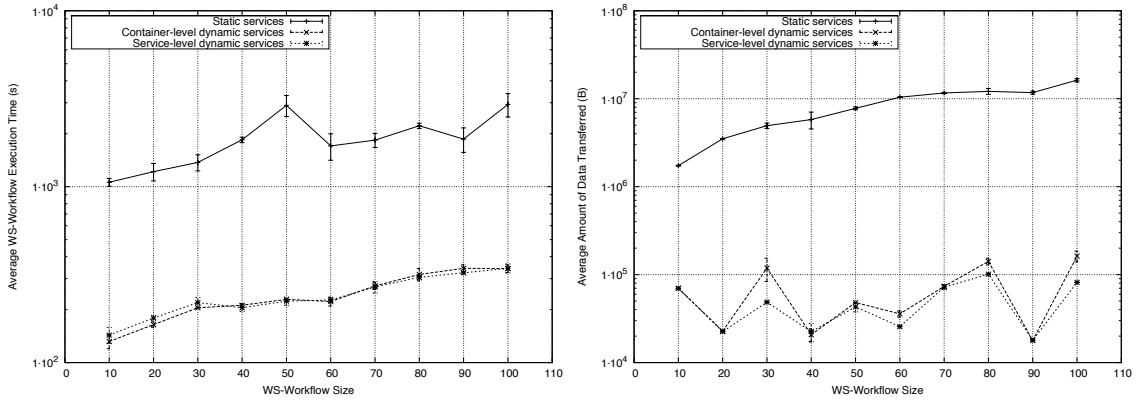
File ID	File Size	File ID	File Size
0	8 KB	5	10 MB
1	100 KB	6	50 MB
2	500 KB	7	100 MB
3	1 MB	8	500 MB
4	5 MB	9	1 GB

Figure 2: Data-intensive workflow

data-intensive WS-Workflow by the size x, we mean $m \times n = x$. The WS-Orc creates a thread every t seconds to process a batch. Each batch is a sequence of Web Service invocations and configured with a randomly generated *configuration string* which specifies the location and size of input files (e.g., $\{(Location, FileId)\} = \{(8,6),(0,1),...,(7,9)\}$ specifies that the first service should be invoked with the URI of the 50 MB input file which is located on compute-node 8 and the last service should be invoked with the URI of the 1 GB input file which is located on compute-node 7). For static services, the data-intensive service is deployed on all compute-nodes and the WS-Orc chooses a random service and invokes it. For dynamic services, no data-intensive service is initially deployed. The WS-Orc sends a `schedule()` request to the WS-Scheduler, and the WS-Scheduler uses the input data location aware algorithm to schedule dynamic deployment of data-intensive Web Service.

4.3.1. Comparison of Static and Dynamic Deployment

We measured the time required to execute concrete and abstract data-intensive WS-Workflows to show the effectiveness of the WS-Scheduler in executing data-intensive WS-Workflows. We executed WS-Workflows with various sizes from 10 to 100 and measured their execution time when using static services and container-level dynamic services. The configuration string of batches are maintained the same for every size of WS-Workflow (e.g., the WS-Workflow with the size 50 has 5 batches; we generated 5 random configuration strings and run the experiments with the same configurations for two service types for a number of times). Throughout these experiments the WAN Model



(a) Average WS-Workflow execution time

(b) Average amount of data transferred

Figure 3: Experiment results for various sizes of data-intensive WS-Workflow

introduced in Section 4.2 is used to generate available bandwidth between nodes and emulate a WAN environment. Figure 3 illustrates the experiment results in the logarithmic scale for the y-axis.

4.3.2. Discussion

Empirical studies showed that the data-intensive WS-Workflow execution time was decreased when using dynamic services (see Figure 3a). The framework allowed moving the service close to or on the same site as where the input data is located instead of moving the large input data to the service. Empirical studies also showed that the amount of data which has been transferred is reduced when using dynamic services (see Figure 3b).

Having showed that the framework is able to dynamically schedule and deploy Web Services on Grid resources, we argue that it also ensures more flexible and fault-tolerant WS-Workflows. A concrete WS-Workflow is not as flexible and fault-tolerant as an abstract WS-Workflow, because if one or more of its services are not available, the user should intervene to choose another available static service. On the other hand, in abstract WS-Workflows, the framework deploys services dynamically which helps having more flexible and fault-tolerant WS-Workflows.

5. Conclusion and Future Work

We proposed, designed and implemented a dynamic Web Service scheduling and deployment framework called DynaSched that uses some of the GT4 services and provides a number of additional services to enable dynamic Web Service scheduling and deployment on the Grid. In brief, DynaSched includes the WS-Orc which uses the WS-Scheduler to execute abstract WS-Workflows on the Grid. The WS-Scheduler decides where to deploy Web Services based on a scheduling algorithm. Currently there is an input data location aware algorithm which is suitable for data-intensive services is implemented. The WS-Scheduler contacts other components of the framework including the WS-CodeR and the WS-Factories along with the information and data management services provided by GT4 to deploy Web Services dynamically on selected compute-nodes.

We performed a series of experiments for data-intensive WS-Workflows to evaluate the performance of the WS-Scheduler and the framework. Empirical studies showed that the framework decreased WS-Workflow execution time. We argued that the framework ensures more flexible, fault-tolerant workflows.

DynaSched is not yet a complete framework. Our research to date focuses on the development of a dynamic Web Service scheduling and deployment framework for Grid WS-Workflows and the algorithms under the framework to support input data location aware scheduling of dynamic Web Services. Currently we are developing an algorithm to support load balancing of dynamic Web Services which is suitable for compute-intensive WS-Workflows. Further research and implementation work along the following directions could also be carried out:

- *Hierarchical WS-Workflow Schedulers:* Building a hierarchy of WS-Schedulers can help utilizing more resources on the Grid and boost the performance of the framework. A meta-scheduler which is able to partition a WS-Workflow into several sub WS-Workflows is a key component to realize a hierarchy of WS-Schedulers.

- *Scheduling Algorithms:* Further scheduling algorithms can lead to a better framework performance for a wider range of WS-Workflows. A number of possible scheduling algorithms are: (i) a hybrid scheduling algorithm for

hybrid WS-Workflows which are composed of both compute-intensive and data-intensive Web Services, (ii) a partitioning algorithm which takes the whole WS-Workflow into consideration and makes scheduling decisions according to the dependencies and logical paths between its Web Services, (iii) customized algorithms which are optimized for real-world WS-Workflows.

- *Standard WS-Workflow Orchestration Engine:* Using a standard WS-Workflow orchestration engine accelerates WS-Workflow composition and facilitates collaboration between users. Research issues to propose and standardize the support of abstract WS-Workflows based on standard workflow specifications such as Web Services Business Process Execution Language (WS-BPEL) should be addressed to achieve this goal.

Acknowledgment

The project was funded by Agency for Science, Technology and Research (A*STAR) and Nanyang Technological University (NTU). DynaSched was implemented and evaluated in the Parallel and Distributed Computing Center (PDCC), School of Computer Engineering, NTU. We would like to extend our appreciation to our colleagues in PDCC and the open source community members whose contributions facilitated the development of DynaSched.

References

[1] The Globus Alliance, An ecosystem of grid components.
URL http://www.globus.org/grid_software/ecology.php
[2] Channabasavaiah, Holley, Tuggle, Migrating to a service-oriented architecture, Tech. rep., IBM DeveloperWorks (2003).
[3] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid - enabling scalable virtual organizations (2001).
URL http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0103025
[4] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration (2002).
[5] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, J. V. Reich, The Open Grid Services Architecture, Version 1.5, Published by Global Grid Forum Open Grid Service Architecture Working Group (July 2006).
URL http://www.ggf.org/documents/GFD.80.pdf
[6] OASIS Web Services Resource Framework (WSRF) Technical Commitee, Web Service Resource Framework.
URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf
[7] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, W. Vambenepe, The WS-Resource Framework Version 1.0 (May 2004).
URL http://www.globus.org/wsrf/specs/ws-wsrf.pdf
[8] I. Foster, Globus toolkit version 4: Software for service-oriented systems, in: IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, 2005, pp. 2–13.
[9] U. o. W.-M. Condor Team, Condor project.
URL http://www.cs.wisc.edu/condor/
[10] Pegasus.
URL http://pegasus.isi.edu/
[11] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, I. Wang, Programming scientific and distributed workflow with Triana services: Research articles, Concurr. Comput. : Pract. Exper. 18 (10) (2006) 1021–1037. doi:http://dx.doi.org/10.1002/cpe.v18:10.
[12] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, T. Oinn, Taverna: a tool for building and running workflows of services., Nucleic Acids Research 34 (Web Server issue) (2006) 729–732.
URL http://view.ncbi.nlm.nih.gov/pubmed/16845108
[13] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludscher, S. Mock, Kepler: An extensible system for design and execution of scientific workflows, in: In SSDBM, 2004, pp. 21–23.
[14] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, M. Wieczorek, ASKALON: A development and grid computing environment for scientific workflows, in: Workflows for e-Science, Springer London, 2007, pp. 450–471. doi:10.1007/978-1-84628-757-2_27.
URL http://www.springerlink.com/content/v8680m853171r462/
[15] M. Keidl, S. Seltzsam, A. Kemper, Reliable web service execution and deployment in dynamic environments, in: Technologies for E-Services, 2003, pp. 104–118.
[16] W3C Commitee, Web services architecture, W3C Working Group Note (February 2004).
URL http://www.w3.org/TR/ws-arch/
[17] L. Qi, H. Jin, I. Foster, J. Gawor, Hand: Highly available dynamic deployment infrastructure for globus toolkit 4, Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on (2007) 155–162doi:10.1109/PDP.2007.49.
[18] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, Grid information services for distributed resource sharing.
[19] University of California, Berkeley Millennium Project, Ganglia.
URL http://ganglia.info/
[20] Enhanced TTCP.
URL http://sourceforge.net/projects/ettcp
[21] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, S.-J. Lee, S3: a scalable sensing service for monitoring large networked systems, in: INM '06: Proceedings of the 2006 SIGCOMM workshop on Internet network management, ACM, New York, NY, USA, 2006, pp. 71–76. doi:http://doi.acm.org/10.1145/1162638.1162650.
[22] Planetlab.
URL http://planet-lab.org/